Katta G. Murty

Indian Statistical Institute

Dura W. Sweeney †

International Business Machines Corporation

Caroline Karel

Case Institute of Technology

AN ALGORITHM FOR THE TRAVELING SALESMAN PROBLEM

John D. C. Little

Massachusetts Institute of Technology

Katta G. Murty [*]

Indian Statistical Institute

Dura W. Sweeney [†]

International Business Machines Corporation

Caroline Karel

Case Institute of Technology

March 1, 1963

[*] Work done while on a study assignment at Case Institute of Technology

[†] Work done while a Sloan Fellow at M. I. T.

Abstract

A "branch and bound" algorithm is presented for solving the traveling salesman problem. The set of all tours (feasible solutions) is broken up into increasingly small subsets by a procedure called branching. For each subset a lower bound on the length of the tours therein is calculated. Eventually, a subset is found which contains a single tour whose length is less than or equal to some lower bound for every tour. The motivation of the branching and the calculation of the lower bounds are based on ideas frequently used in solving assignment problems.

Computationally, the algorithm extends the size of problem which can reasonably be solved (i.e. the size of problem for which a guaranteed optimal tour can be found without using methods special to the particular problem). The algorithm has been programmed for an IBM 7090. Problems constructed by using 3 digit random numbers as distances between cities have required average computing times as follows: 10 cities, .012 minutes; 20 cities, .084 minutes; 30 cities, .98 minutes; 40 cities, 8.37 minutes. Because the time is growing roughly exponentially, the solution of problems much larger appears impractical under the current algorithm. Symmetric distance matrices (e.g. from a map) take longer than the asymmetric random distance matrices. For example, Held and Karp's 25 city problem took 4.7 minutes. For both types of problems the computing times display considerable variance.

Acknowledgement

The computing time used in the early stages of the work was provided by the Case Institute of Technology Computing Center, the M. I. T. Computation Center, and the M. I. T. School of Industrial Management Computing Facility. The time for the production runs was provided by I. B. M. on the IBM 7090 at the M. I. T. Computation Center. We wish to acknowledge the help of R. H. Gonzalez for programming certain investigations. His time was supported by the U. S. Army Research Office (Durham).

Introduction

The traveling salesman problem is easy to state: A salesman, starting in one city, wishes to visit each of n-1 other cities and return to the start. In what order should he visit the cities to minimize the total distance traveled? For "distance" we can substitute time, cost, or other measure of effectiveness as desired. Distance or costs between all city pairs are presumed known.

The problem has become famous because it combines ease of statement with difficulty of solution. The difficulty is entirely computational, since a solution obviously exists. There are (n-1)! possible tours, one or more of which must give minimum cost. (The minimum cost could conceivably be infinite - it is conventional to assign an infinite cost to travel between city pairs which have no direct connection.)

The traveling salesman problem recently achieved national prominence when a soap company used a 33 city example as the basis of a promotional contest. Quite a few people found the best tour. A number of people (mostly from universities) wrote the company that the problem was impossible - an interesting misinterpretation of the state of the art.

For the early history of the problem, see Flood[1]. In recent years a number of methods for solving the problem have been put forward. Some suffer from computational inefficiency, others lack guarantees that the solution will be optimal, and still others require intuitive judgments that would be hard to program on a computer. For a detailed discussion, see Gonzalez[2].

The approach which, to date, has been pursued furthest computationally is that of dynamic programming. Held and Karp[3] and Gonzalez[2] have independently applied the method and have solved various test problems on computers. Gonzalez programmed an IBM 1620 to handle problems up to 10 cities. In his work the time to solve a problem grew somewhat faster than exponentially as the number of cities increased. A 5 city problem took 10 seconds, a 10 city problem took 8 minutes, and the addition of one more city multiplied the time by a factor, which, by 10 cities, had grown to 3. Storage requirements expanded with similar rapidity.

Held and Karp[3] have solved problems up to 13 cities by dynamic programming using an IBM 7090. A 13 city problem required 17 seconds. But such is the power of an exponential that, if their computation grows at the same rate as that of Gonzalez, a 20 city problem would require about 10 hours. Storage requirements, however, may become prohibitive before then. For larger problems than 13 cities, Held and Karp develop an approximation method which seems to work well but does not guarantee an optimal tour. Our concern here will be with methods that must eventually yield an optimal tour.

We have found two papers in which the problem has been approached by methods similar to our "branch and bound" algorithm. Rossman, Twery and Stone[4] in an unpublished paper apply ideas which they have called combinatorial programming[5]. To illustrate their method they present

a 13 city problem. It was solved in 8 man-days. We have solved their problem by hand in about 3 1/2 hours. Eastman[6] in an unpublished doctoral thesis and laboratory report presents a method of solution and several variations on it. His work and ours contain strong similarities. However, to use our terminology, his ways of choosing branches and of calculating bounds are different from ours. He basically solves a sequence of assignment problems which give his bounds. We have a simpler method, and for branching we use a device which has quite a different motivation. The biggest problem Eastman solves is 10 cities and he gives no computation times, so that effective comparisons are difficult to make.

To summarize, the largest problem which we know about that has been solved by a general method which guarantees optimality and which can reasonably be programmed for a computer is 13 cities. Our method appreciably increases this number. However, the time required increases at least exponentially with the number of cities and eventually, of course, becomes prohibitive. Detailed results are given below.

The Algorithm

We shall simultaneously present the algorithm and work out an example. Heuristic arguments accompany each step. In the next section a more detailed mathematical justification will be given.

Consider the matrix of Figure 1. The entry at position (i,j), say c(i,j), represents the cost (distance) for going from city i to city j. A tour is a set of city pairs, e.g.,

$$t = [(1,3)\ (3,2)\ (2,5)\ (5,6)\ (6,4)\ (4,1)]$$

which spell out a trip that goes to each city once and only once. Let z be the cost of a tour. From Figure 1 it may be seen that the above tour would cost:

$$z = 43 + 13 + 30 + 5 + 9 + 21 = 121.$$

If a constant is subtracted from each element of the first row of Figure 1, that constant is subtracted from the cost of every tour. This is because every tour must include one and only one element from the first row. The relative costs of all tours, however, are unchanged and so the tour which would be optimal is unchanged. The same argument can be applied to the columns.

The process of subtracting the smallest element of a row from each element of a row will be called reducing the row. Thus the first row in Figure 1 can be reduced by 16. Note that, in terms of the unreduced matrix, every trip out of city 1 (and therefore every tour) will have a cost of at least 16. Thus, the amount of the reduction constitutes a lower bound on the length of all tours in the original matrix.

Step 1: Reduce the rows and columns of the cost matrix. Save the sum of the reductions as a lower bound on the cost of a tour.

	1	2	3	4	5	6
1	∞	27	43	16	30	26
2	7	∞	16	1	30	25
3	20	13	∞	35	5	0
4	21	16	25	∞	18	18
5	12	46	27	48	∞	5
6	23	5	5	9	5	∞

Figure 1. Cost matrix for a 6-city problem.

	1	2	3	4	5	6
1	∞	11	27	⑩ 0	14	10
2	1	∞	15	① 0	29	24
3	15	13	∞	35	5	⑤ 0
4	① 0	⓪ 0	9	∞	2	2
5	2	41	22	43	∞	② 0
6	13	⓪ 0	⑨ 0	4	② 0	∞

Figure 2. Cost matrix after row and column reduction.

The results of reducing Figure 1 are shown in Figure 2. The total reduction is 48 and so $z \geq 48$ for all tours.

Next we split the set of all tours into two disjoint subsets. This is conveniently indicated by drawing a tree as in Figure 3. The node (branching point) containing "all tours" is self-explanatory. The node containing 1,4 represents all tours which include the city pair (1,4). The node containing $\overline{1,4}$ represents all tours which do not. From the 1,4 node we might later want to branch again, say, on the basis of (2,1). In Figure 3 the node containing $\overline{2,1}$ represents all tours which include (1,4) but not (2,1) whereas 2,1 represents all tours which include both (1,4) and (2,1). In general, by tracing back from a node to the start we can pick up which city pairs are specified to be in and which out of the tours represented by the node. If the branching process is carried far enough, some node will eventually represent a single tour. Notice that at any stage of the process , the union of the sets represented by the terminal nodes is the set of all tours.

Returning to the question of lower bounds, we have seen that 48 is a lower bound on all tours. In Figure 2 the starting node is accordingly marked with 48. Consider the first row of Figure 2. Suppose that city pair (1,4) is <u>not</u> in a tour. Then since city 1 must be assigned to <u>some</u> city, the tour must incur a cost of at least 10 (the second smallest element in row 1). This is on top of the 48 already reduced out of the matrix. Similarly, since some city must be assigned to city 4, the tour

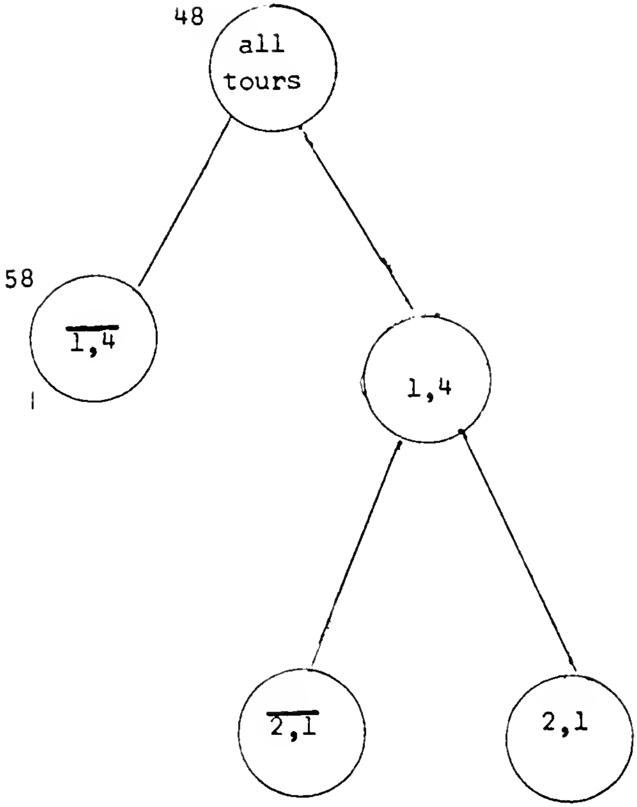

Figure 3. Start of tree.

	1	2	3	4	5	6
1	~~∞~~	~~11~~	~~27~~	~~0~~	~~14~~	~~10~~
2	0 ~~1~~	∞	14 ~~15~~	0	28 ~~29~~	23 ~~24~~
3	15	13	∞	35	5	0
4	∞	0	9	∞	2	2
5	2	41	22	43	∞	0
6	13	0	0	4	0	∞

Figure 4. Matrix after deletion of row 1 and column 4.

must further incur the cost of the second smallest element in column 4. The element happens to be zero in this case and so the total lower bound is still 58. The node has been so marked in Figure 3.

Thinking ahead, suppose we eventually discover a tour with z = 56. Then, in a search for the optimal tour, it would be unnecessary to examine any of the tours in the node $\overline{1,4}$ since they all have $z \geq 58$. Not knowing in advance what z's will be found, we shall select the element on which to branch so as to increase most of this lower bound. Let $\theta(k,\ell)$ be the jump in lower bound if (k,ℓ) is chosen for branching. As indicated in the example:

$$\begin{aligned}\theta(k,\ell) &= \text{smallest element in row } k\text{, omitting the } (k,\ell) \text{ element}\\ &\quad + \text{smallest element in column } \ell\text{, omitting the } (k,\ell) \text{ element}\end{aligned}$$

Step 2: Given the node, say X, from which to branch next, and given the cost matrix associated with X, find the city pair (k,ℓ) which maximizes θ and extend the tree from X to a node $\overline{k,\ell}$. Add $\theta(k,\ell)$ to the lower bound of X to set the lower bound of the new node.

Inspection of the matrix will show that $\theta(k,\ell) = 0$ unless $c(k,\ell) = 0$ so that the search for max θ is confined to the zeros of the matrix. In Figure 2 the θ values are shown in small circles. The largest occurs for (1,4), as already chosen (clairvoyantly) to illustrate branching.

At this point we slip in a step needed later to detect when the end of a tour is near.

Step 3: If the number of city pairs committed to the tours of X is n-2, go to Step 6. Otherwise continue.

Returning to the example, we next seek a lower bound for the 1,4 node. Since all tours here contain (1,4) row 1 and column 4 are no longer needed andmay be crossed out. Next observe that no tour in the set represented by this node can contain the city pair (4,1). This is because (1,4) (4,1) constitutes a two city subtour, and it is impossible for a tour to contain a subtour. Therefore, without loss of tours, write 1,4 and $\overline{4,1}$ in the node and in the cost matrix set $c(4,1) = \infty$. This will prevent (4,1) from being chosen as part of a possible tour. After the changes in the matrix, further reduction of rows or columns may be possible. In the case at hand, row 2 can be reduced by one. See Figure 4. Adding this to the previous lower bound gives 49, which has been marked on the node in Figure 5.

Step 4: Finish the branching based on (k,ℓ) by extending the tree from X to a node $k,\ell;\overline{m,p}$. Here (m,p) is the city pair which would join the ends of the longest connected path involving (k,ℓ) in the set of committed city pairs of the new node. Delete row k and column ℓ in the cost matrix and set $c(m,p) = \infty$. Reduce rows and columns if possible and add the amount of the reduction to the lower bound of X to set the lower bound for the new node.

A further example of finding (m,p) is illustrated by the node $2,1;\ \overline{4,2}$. All tours at this node contain (2,1) and (1,4). The addition of (4,2) would create a 3 city subtour, which we wish to prevent and do

prevent by specifying $\overline{4,2}$ for the node. One could also add $\overline{1,2}$ but row 1 has already been crossed out and so there is no advantage. The only worthwhile candidate for (m,p) is always the element that completes the longest subtour involving (k,ℓ).

We now repeat the basic branching procedure of Steps 2 and 4; extending the tree, and working up the lower bounds as we go.

Step 5: If no node has been found which contains only a single tour, find the terminal node with the smallest lower bound and return to Step 2 for branching. Otherwise continue.

A comment is required about branching to the right versus branching to the left. Branching on the k,ℓ; $\overline{m,p}$ node (to the right in Figure 5) is the usual operation. It involves crossing out rows and columns and other manipulations which are conveniently done on the same physical representation of the matrix, whether stored on paper or in a computer. When one returns to extend a $\overline{k,\ell}$ node (moving to the left) it is usually advantagous to reconstruct an appropriate matrix from the original cost matrix, rather than arranging to save cost matrices for each node as it is laid out. We, therefore, give

Step 2a: If a cost matrix is needed for node X, start with the original cost matrix and

(1) Find the city pairs committed to be in the tours of X and add their cost elements together to form part of the lower bound for X.

(2) For every (i,j) which is one of these city pairs, cross out the i^{th} row and j^{th} column of the matrix. Add infinities at prohibited city pairs.

(3) Reduce the remaining matrix and add the amount of the reductions to that found in (1) to give a lower bound for X.

(4) Perform the instructions given in Step 2.

The lower bound and the reduced matrix are not necessarily unique, consequently, when a new bound is calculated, the old is discarded. However, bounds calculated in the two ways will usually be the same.

As Steps 1-5 are performed on the example, the tree of Figure 5 is developed, up to the final node: 4,3; 6,2.

Here, Step 3 causes a jump out of the loop. Step 2 will have told us that the current branching is based on (4,3). This, plus the city pairs found by tracing back the tree, show that n-1 city pairs are already committed to the upcoming node. But the n-1 determine what the last must be (here (6,2)), and we have found a single tour node. Examination of the final 2X2 cost matrix will show that no reduction is possible and that the cost of the remaining pairs relative to this matrix is zero. Thus the tour has z = 63.

Furthermore, since every possible tour is contained in one or another of the terminal nodes and each of the other nodes has a lower bound greater then 63, we have found the optimal tour.

Step 6: If entry to this step is from Step 3, the next node is $k,\ell;\ m,p$ where (m,p) is the only city pair left after crossing out row k

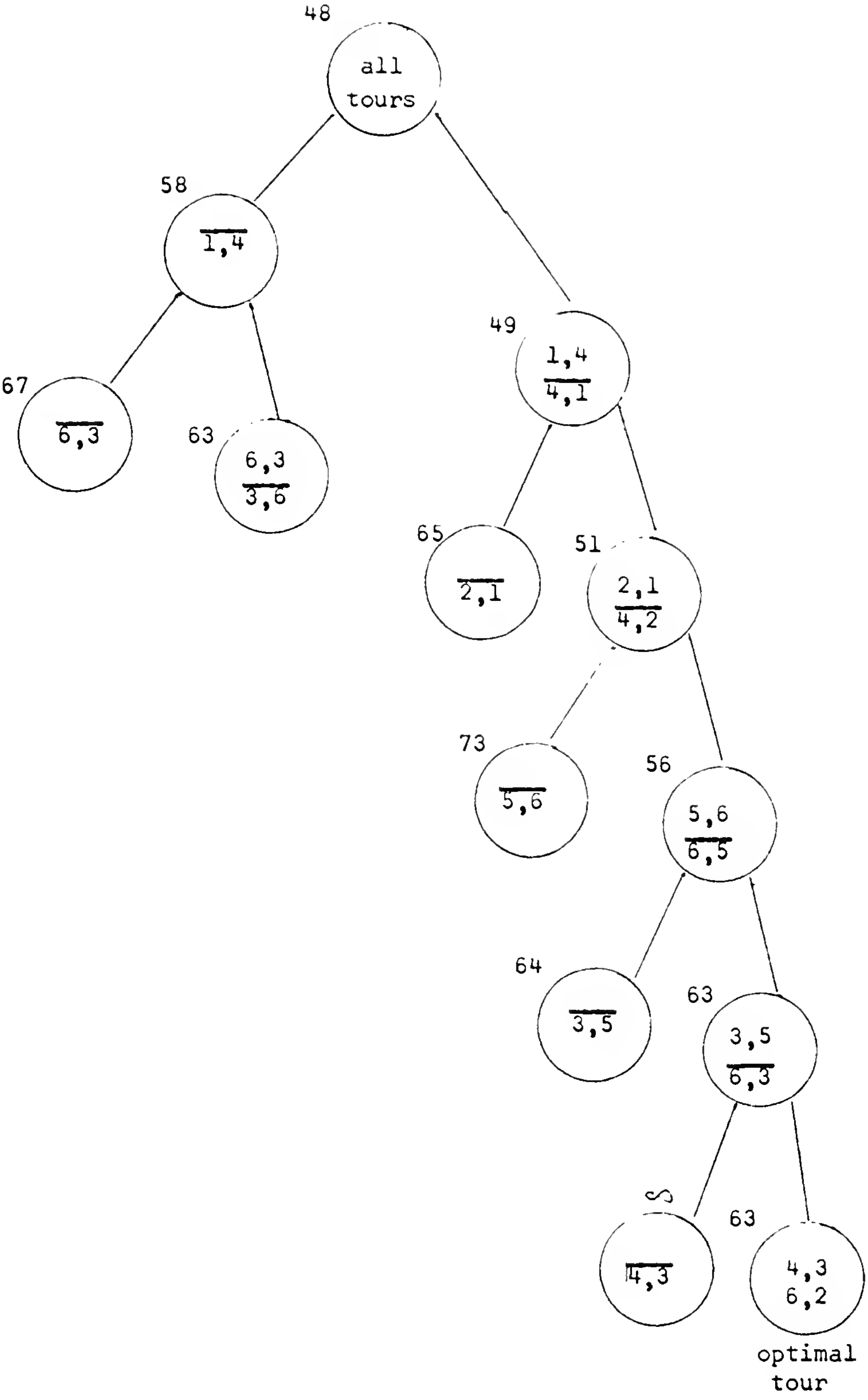

Figure 5. Final Tree.

and column ℓ. The node contains a single tour. Calculate its cost. Now, for any entry to the step, let t_0 denote the tour with the smallest cost, say z_0, among single tours found so far. If z_0 is less than or equal to the lower bound of every terminal node on the tree, t_0 is optimal. Otherwise choose the multiple tour terminal node with the smallest lower bound and return to Step 2 for branching.

At this point, let us stand back and review the general motivation of the algorithm. It proceeds by branching, crossing out a row and column, blocking a subtour, reducing the cost matrix to set a lower bound and then repeating. Although it is fairly clear that the optimal solution will eventually be found, why should these particular steps be expected to be efficient? First of all, the reduction procedure is an efficient way of building up lower bounds and also of evoking likely city pairs to put into the tour. Branching is done so as to maximize the lower bound on the $\overline{k,\ell}$ node without worrying about the k,ℓ node. The reasoning here is that the k,ℓ node represents a smaller problem, one with the k^{th} row and ℓ^{th} column crossed out. By putting the emphasis on a large lower bound for the larger problem, we rule out non-optimal tours faster.

Insight into the operation of the algorithm is gained by observing that the crossing out of a row and column and the blocking of the corresponding subtour creates a new traveling salesman problem having one fewer city. Using the notation of step 4, we can think of city m and city p as coalesced into a single city, say, m'. Setting $c(m,p) = \infty$

is the same as setting $c(m^t, m^+) = \infty$. The blocking of subtours is a way of introducing the tour restrictions into what is essentially an assignment problem and is accomplished rather successfully by the algorithm.

Finally, unlike most mathematical programming algorithms, the one here has an extensive memory. It is not required that a trial solution at any stage be converted into a new and better trial solution at the next stage. A trial branch can be dropped for a moment while another branch is investigated. For this reason there is considerable room for experiment in how the next branch is chosen. On the other hand the same property leads to the ultimate demise of the computation - for *n* sufficiently large there are just too many branches to investigate and a small increase in *n* is likely to lead to a large number of new nodes that require investigation.

Mathematical Support

We shall establish the convergence of the algorithm, the optimality of the final result, the justification of the branching procedure, and the validity of the lower bound calculations. First a formal problem statement is presented.

Problem Statement

Index the cities of the problem by i=1, ..., n. The couple (i,j) is an ordered *city pair* in which *i* is an origin and *j* is a destination. A *tour*, *t*, is a set of *n* city pairs,

$$t = [(i_1, j_1)\ (i_2, j_2)\ \dots\ (i_n, j_n)]$$

satisfying the constraints:

C1. Each integer from 1 to _n_ must appear once and only once as an origin and once and only once as a destination.

C2. It must be possible to arrange the city pairs into the form

$$t = [(i_1,i_2)\ (i_2,i_3)\ \ldots\ (i_{n-1},i_n)\ (i_n,i_1)]$$

Thus the elements of _t_ are arcs in an elementary circuit of length _n_. Circuits of length less than _n_ are _subtours._ Note that _t_ cannot be a tour and also contain a subtour; for there is no way to connect the necessary cities to the subtour and still meet the constraits.

Let $c(i,j)$ = the cost of going from _i_ to _j_.

C = the _n_ x _n_ of the $c(i,j)$

$z(t) = \sum_{(i,j)\in t} c(i,j)$ = the cost of tour _t_

T = the set of all tours

Formally, then, the traveling salesman problem is to choose $t \in T$ to minimize $z(t)$.

Remarks: Traveling salesman problems are frequently symmetric, i.e., $c(i,j) = c(j,i)$, and a problem involving straight line distances on the Euclidian plane would require various geometric conditions to be satisfied. The algorithm here does not require either of these properties. However, if the problem is symmetric, a modification of the method makes it possible to take some advantage from the fact.

If, for some reason, it is not possible to go from i to j, we set $c(i,j) = \infty$. Thus an infinitely expensive tour is an infeasible one.

Observe that if constraint C2 is dropped, _t_ becomes an _assignment_ and the problem statement becomes that of the assignment problem.

Branch and Bound

The idea that we are calling "branch and bound" is more general than the traveling salesman algorithm. It may be summarized as follows:

A minimal solution for a problem can be found by taking the set of all feasible solutions and breaking it up into disjoint subsets. For each subset a lower bound is found on the value (of the objective function) for the minimal solution in that subset. The subset with the smallest lower bound is broken up further and lower bounds are again found for the new subsets. The process continues this way and is conveniently visualized as a branching process, or tree. The branching points (nodes) represent the subsets which have been broken up and end points (terminal nodes) represent the current collection of subsets. The process stops when a subset is found which contains a single solution whose value is less than or equal to the lower bounds for all the terminal nodes. (The lower bound used for a single solution is assumed to be the value of the solution itself.) Since the terminal nodes collected together always contain all solutions, a minimal solution has been identified.

The process is sure to converge if, for example, the number of solutions is finite and if the partitioning procedure will separate out every solution in a finite number of steps. At worst, the procedure will simply become an enumeration of solutions. The algorithm's efficiency, or lack of it, will rest on the devices used to break up the subsets and find the lower bounds.

In the traveling salesman problem, the number of solutions is finite

and, as will be seen in the next section, the branching procedure would eventually enumerate all solutions if not stopped by finding the optimal solution sooner. Thus, convergence will be assured and, by the arguments above, the final result will be optimal.

Justification of the Branching Procedure

Several proporties of the branching procedure are to be verfied: (1) The process leads to single tour subsets; (2) the process converges in a finite number of steps, and (3) at any stage the union of the sets of the terminal nodes is T.

As a result of Steps 2 and 4, each branching splits a subset of tours, X, into two disjoint subsets, say X(1) and X(2), on the basis of some city pair (k,ℓ):

$$X(1) = \{t \in X \mid (k,\ell) \notin t\}$$
$$X(2) = \{t \in X \mid (k,\ell) \in t,\ (m,p) \notin t\}$$

except in the case covered by Step 3, in which situation we claim that $X(2) = \{t \in X \mid (k,\ell) \in t,\ (m,p) \in t\}$. In any case, by tracing back up the tree to T from a node X, we can always develop two sets of city pairs:

$a[X] = [(i_1,j_1)\ (i_2,j_2)\ldots]$ = a set of city pairs committed to appear in $t \in X$

$b[X] = [(r_1,s_1)\ (r_2,s_2)\ldots]$ = a set of city pairs prohibited from appearing in $t \in X$.

The properties to be established follow easily from the following:

Theorem 1: Under the algorithm, the set a[X] never contains a subtour. Furthermore, the choice of (m,p) never prohibits a tour in X(2)

that would otherwise be there. If Step 6 is entered from Step 3, X(2) contains the single tour $t = a[X] \cup \{(k,\ell)\ (m,p)\}$.

In Step 4, $a[X(2)] = a[X] \cup \{(k,\ell)\}$. Has the addition of (k,ℓ) produced any new paths in a[X(2)] which could be made into circuits (subtours) at the next branching? Yes, (k,ℓ) will either extend a path of a[X], join two paths of a[X], or start a new, single arc path. In any of these cases the next branching might select a city pair which would close the extended or new path. But whichever is the case, the circuit is then blocked in advance by $b[X(2)] = b[X] \cup \{(m,p)\}$. Only the potential circuits newly created by (k,ℓ) need be considered, for others have been blocked by the same procedure applied at an earlier stage. Circuits which close by joining into the middle of a path will never be introduced because the row and column for such an entry point has been crossed out of the matrix. Therefore, a[X] never contains a subtour. Furthermore, under the algorithm no tours are lost by prohibiting (m,p) in tours of X(2), since, starting from (m,p) we can form no tour which includes all of the rest of a[X(2)]. A potential exception is prevented by Step 3. If Step 6 is entered from Step 3, $a[X] \cup \{k,\ell\}$ contains (n-1) elements and the n^{th} is uniquely determined. Since subtours have been prohibited, the result is a tour, and obviously a single tour. However, the definition of (m,p) includes the possibility that it completes a tour, and so the n^{th} element is (m,p). This finishes the theorem.

As we follow down the tree, the sets a or b or both are augmented at each branching. Eventually, if not otherwise stopped, we always reach

(in a finite number of stages) a node which has *n* elements in *a* or else a node which has so many elements in *b* as to prohibit all tours. This last will produce an infinite lower bound, thereby shifting the branching elsewhere. Thus the process will converge, and, as indicated by the theorem, will produce single tour subsets.

Since branching loses no tours: $X(1) \cup X(2) = X$ for every branch point. The union of the terminal nodes is T at every stage.

Validity of the Lower Bounds

The calculation of the lower bounds is built around the concept of reduction and the following well known result.

Theorem 2: Let C and C' be n x n cost matrices and z(t) and z'(t) be the cost of a tour t under C and C' respectively. If C' is formed by subtracting a constant θ, from each element in a row (column) of C then

(a) $z(t) = \theta + z'(t)$ *for all* $t \in T$

(b) *The optimal tours under C' are the same as under C.*

Part (a) is true because a tour must include *some* element (but only one) from each row (or column). Part (b) follows from (a) since relative *z*'s are unchanged.

A *reduced matrix* will be defined as a matrix of non-negative elements having at least one zero in each row and column. Any matrix can be reduced by subtracting from each row its row minimum and then from each column its column minimum, but sometimes more complicated procedures are desirable. The sum of the constants used in the subtractions will be called the *amount of reduction* of the matrix. Neither the reduced matrix

nor the amount of reduction is necessarily unique but may depend on the way it is performed.

Corollary 1: If C is a matrix, C' a reduced matrix obtained from C, and θ the amount of reduction, then (a) and (b) of Theorem 2 hold. Furthermore $z(t) \geq \theta$ for all $t \in T$.

Results (a) and (b) follow immediately from Theorem 2. That θ is a lower bound follows from (a) and the non-negativity of a reduced matrix.

The proofs of Theorem 2 and corollary 1 are unchanged if t is taken to be an assignment and T the set of all assignments.

Let $C[X]$ = a reduced matrix which lacks row $\underline{i}$ and column $\underline{j}$
for each $(i,j) \in a[X]$

$c_X(i,j)$ = the i,j element of $C[X]$

$t-a$ = the set of city pairs which are in $t \in X$ but not in $a[X]$

$z(t-a)$ = the cost of the assignment $t-a$ under $C[X]$

$w[X]$ = a lower bound on $z(t)$, $t \in X$.

We shall show that the algorithm sets up and preserves a "node condition", whereby w[X] and an associated C[X] are known and possess properties which enable each lower bound to be determined from a previous one.

Node Condition: Associated with any node X is a lower bound w[X] and a reduced matrix C[X] such that for all $t \in X$

$$z(t) = w[X] + z(t-a)$$

Theorem 3: (a) Step 1 leaves the node condition satisfied for X=T.

(b) Step 2a leaves the node condition satisfied for X.

(c) If X satisfies the node condition, then after Step 2, X(1) satisfies it.

(d) If X satisfies the node condition, then after Step 4, X(2) satisfies it.

Part (a) is immediate from corollary 1.

For (b), let

$$\phi[X] = \sum_{(i,j)\in a} c(i,j)$$

C[X] = the matrix formed in Step 2a.

$\theta[X]$ = the amount of the reduction in part (3) of Step 2a.

For $t \in X$

$$z(t) = \phi[X] + \sum_{(i,j)\in t-a} c(i,j)$$

Let C" with elements c"(i,j) be the matrix formed by performing (2) of Step 2a. Since the infinite elements occur only for $(i,j) \notin t$,

$$z(t) = \phi[X] + \sum_{(i,j)\in t-a} c''(i,j)$$

Applying Corollary 1 to the assignments of C" and C[X]:

$$\sum_{(i,j)\in t-a} c''(i,j) = \theta[X] + z(t-a)$$

whence

$$z(t) = \phi[X] + \theta[X] + z(t-a)$$

Step 2a sets $w[X] = \phi[X] + \theta[X]$. This is a lower bound because z(t-a) is non-negative. The node condition is satisfied.

For (c), Step 2 implies that C[X(1)] can be obtained by taking C[X], replacing $c_X(k,\ell)$ by infinity, and reducing.

For $t \in X$ we know

$$z(t) = w[X] + z(t-a)$$

where $z(t-a)$ refers to $C[X]$. After insertion of the infinity, the cost of $(t-a)$ under the new matrix is still $z(t-a)$ for $t \in X(1)$. Reduction of the k^{th} row and ℓ^{th} column yields $C[X(1)]$ and as the amount of reduction, $\theta(k,\ell)$. Let $z_1(t-a)$ denote the cost of $(t-a)$ under $C[X(1)]$. By corollary 1

$$z(t-a) = \theta(k,\ell) + z_1(t-a)$$

Substituting above and setting $w[X(1)] = w[X] + \theta(k,\ell)$ as is done in Step 2, we find that the node condition is satisfied.

For part (d),

$$z(t) = w[X] + z(t-a)$$

where $z(t-a)$ is under $C[X]$.

Furthermore, $z(t-a) = c_X(k,\ell) + \theta_2 + z(t-a[X(2)])$, where $c_X(k,\ell) = \upsilon$, θ_2 is the reduction found in Step 4 and the last term is the cost of the indicated assignment under the $C[X(2)]$ developed in Step 4. Step 4 sets $w[X(2)] \mp w[X] + \theta_2$ and the node condition is satisfied.

The following simplifies the cost calculation in Step 6:

<u>Theorem 4: If $a[X]$ contains n-2 elements and t is the resulting single tour in $X(2)$, then $z(t) = w[X]$.</u>

We know $t = a[X] \cup \{(k,\ell)\ (m,p)\}$ and from the node condition for X:

$$z(t) = w[X] + z(t-a).$$

But $$z(t-a) = c_X(k,\ell) + c_X(m,p)$$

C[X] is a 2 x 2 matrix with a zero in each row and column. One of these must be $c_X(k,.\ell)$. The element $c_X(m,p)$, which is diagonal to $c_X(k,\ell)$, must also be zero because, if not, all elements but $c_X(m,p)$ must be zero. But then $\theta(k,\ell) = 0$ and at the same time $\theta > 0$ for two other zeros. But this contradicts the fact that (k,ℓ) produced max θ. Therefore

$$c_X(k,\ell) = c_X(m,p) = 0 \text{ and } z(t) = w[X].$$

In computation, an infinity is represented by some large number. If a[X] and b[X] are inconsistent, X is void, a large number will be reduced out of the cost matrix, and the node will not be examined further (unless possibly the problem has no finite solution, in which case all nodes would eventually have large w[X]. The computer can be programmed to detect this, if desired.) A large negative z(t) would indicate an unbounded minimum.

Modifications

A variety of embellishments on the basic method can be proposed. We record several that are incorporated in the computer program used in later calculations.

Go To The Right

It is computationally advantageous to keep branching to the right until it becomes obviously unwise. Specifically, the program always branches from the node $k,\ell;\overline{m,p}$ unless the lower bound for the node exceeds or equals the cost of a known tour. As a result a few extra nodes may be examined, but usually there will be substantial reduction in the number of time-consuming setups of Step 2a.

One consequance of the modification is that the calculation goes directly to a tour at the beginning. Then, if the calculations are stopped before optimality is proven, a good tour is available. Th~~is~~re is also available a lower bound on the optimal tour. The bound may be valuable in deciding whether the tour is sufficiently good for some practical purpose.

Throw Away the Tree

A large problem may involve thousands of nodes and exceed the capacity of high speed storage. A way to save storage at the expense of time is as follows: Proceed by branching to the right (storing each terminal node) until a single tour is found with some cost, say, z_0. Normally, one would next find the terminal node with the smallest lower bound and branch from there. Instead, work back through the terminal

nodes, starting from the single tour, and discard nodes from storage until one is found with a lower bound less than z_0. Then, branch again to the right all the way to a single tour or until the lower bound on some right hand node builds to z_0. (If the branch goes to the end, a better tour may be found and z_0 assigned a new, lower balue.) Repeat the procedure: again work up the branch, discarding terminal nodes with bounds equal or greater than z_0 until the first one smaller is found; again branch to the right, etc.

The effect of the procedure is that very few nodes need be kept in storage - something on the order of a few $\underline{n}$. These form an orderly sequence stretching from the current operating node directly back to the terminal node on the left-most branch out of "all tours".

As an illustration, consider the problem and tree of Figure 5. The computation would proceed by laying out in storage the nodes $\overline{1,4}$; $\overline{2,1}$; $\overline{5,6}$; and $\overline{3,5}$. At the next step we find a tour with $z_0 = 63$ and the obviously useless node $\overline{4,3}$. The tour is stored separately from the tree. Working up the branch, first $\overline{3,5}$ is discarded, then $\overline{5,6}$ and $\overline{2,1}$, but $\overline{1,4}$ has a bound less than z_0. Therefore, branching begins again from there. A node $\overline{6,3}$ is stored and then we find the node to the right has a bound equal z_0 and may be discarded. Working back up the tree again, $\overline{6,3}$ is discarded and, since that was the only remaining terminal node, we are finished.

The procedure saves storage but increases computation time. If the first run to the right turns up a rather poor tour, i.e. large z_0, the

criterion for throwing away nodes is too stiff. we force ourselves to branch out from many nodes whose lower bounds actually exceed the cost of the optimal tour. Our original method would never do this for it would never explore such nodes until it had finished exploring every node with a smaller bound. In the process, the optimal tour would be uncovered and so the nodes with larger bounds would never be examined.

Taking Advantage of Symmetry

If the travelling salesman problem is symmetric and $\underline{t}$ is any tour, another tour with the same cost is obtained by traversing the circuit in the reverse direction. Probably the most promising way to handle this is to treat the city pair (i,j) as not being ordered. This leads naturally to a new and somewhat more powerful couterpart to $\theta(k,\ell)$. Although the basic ideas are not changed much, considerable reprogramming is required. So far, we have not done it.

There is another way to take advantage of symmetry and this one is easy to incorporate into our program. All reverse tours can be prohibited by modifying X(1) whenever a[X] = 0. Before modification

$$X(1) = \{t \in T \mid (k,\ell) \notin t,\ b[X] \cap t = 0\}$$

$$X(2) = \{t \in T \mid (k,\ell) \in t,\ (m,p) \notin t,\ b[X] \cap t = 0\}$$

All reverse tours of X(2) will have the property $(\ell,k) \in t$. They cannot be in X(2) for this would imply a subtour. Such of them as were in X must be in X(1) and we may easily eliminate them by modifying X(1) (whenever a [X] = 0) to

$$X(1) = \{t \in X \mid (k,\ell) \notin t,\ (\ell,k) \notin t\}$$

A Computational Aid

In both hand and machine computation $\theta(k,\ell)$ is easiest calculated by first finding, for each row $\underline{k}$ and column $\underline{\ell}$ of the reduced matrix:

$\alpha(k)$ = the second smallest cost in row $\underline{k}$.

$\beta(\ell)$ = the second smallest cost in column $\underline{\ell}$.

Then $\theta(k,\ell) = \alpha(k) + \beta(\ell)$ for any (k,ℓ) which has $c(k,\ell) = 0$. In a hand computation the $\alpha(k)$ can be written as an extra column to the right of the matrix and the $\beta(\ell)$ as an extra row at the bottom. By working out a few problems, one easily learns that when the branching is to the right there is no need to search the whole matrix to reset α and β. But that only certain rows and columns need be examined.

Calculations

Problems up to 10 cities can be solved easily by hand. Although we have made no special study of the time required for hand computations, our experience is that a 10 city problem can be solved in less than an hour.

The principal testing of the algorithm has been by machine on an IBM 7090. Two types of problems have been studied: (1) asymmetric distance matrices with elements consisting of uniformly distributed 3 digit random numbers and (2) various published problems and subproblems constructed there from by deleting cities. Most of the published problems have been made up from road atlases or maps and are symmetric.

The random distance matrices have the advantage of being statistically well defined. Average computing times are displayed in Table I and curve (a) of Figure 6. Problems up to 20 cities usually require only a few seconds. The time grows exponentially, however, and by 40 cities is beginning to be appreciable, averaging a little over 8 minutes. As a rule of thumb, adding 10 cities to the problem multiplies the time by a factor of 10. The standard deviation of the computing time also increases with problem size as may be seen in Table I. Because the distribution of times is skew, the simple standard deviation is a little misleading, at least for the purpose of estimating the probability of a long calculation. Consequently, a log normal distribution has been fitted to the tail of the distribution. A use of the tabulated numbers would be, for example, as follows: A two sigma deviation on the high side in a 40 city problem would be a calculation which took $(3.74)^2(4.55) =$ 64 minutes. In other words, the probability that a 40 city random

TABLE I

Mean and Standard Deviation of T

for Random Distance Matrices

(T = time in minutes used to solve

Traveling Salesman Problem on IBM 7090)

Number of Cities	Number of Problems Solved	Mean T	Stat.Dev. T	Mean * Log T	Stat. Dev. * Log T
10	100	.012	.007	log .015	log 1.24
20	100	.084	.063	log .067	log 2.09
30	100	.975	1.240	log .63	log 2.94
40	5	8.37	10.2	log 4.55	log 3.74

* Obtained by plotting the cumulative sequence on log normal probability paper and fitting a straight line, except in the 40 cities case for which the computation was numerical. In the case of 10 cities the log normal fits only the tail of the distribution - 30% of the problems went directly to the solution without extra branching and thereby produced a lump of probability at .002 minute.

distance problem will require 64 minutes or more is estimated to be .023.

Symmetric problems have usually taken considerably longer than random distance problems of the same size. To obtain a variety of problems of increasing size, we have taken published problems and abstracted subproblems of increasing size. The first 10 cities were taken, then the first 11 cities, etc., until the computing times become excessive. Curves (b) and (c) of Figure 6 show the results for subproblems pulled out of the 25 and 48 city problems of Held and Karp.[3] The 25 city problem itself took 4.7 minutes. We note that Held and Karp's conjectured optimal solution is correct.

A few miscellaneous problems have also been solved. Croes'[8] 20 city problem took .126 minutes. A 64 "city" knight's tour took .178 minutes.

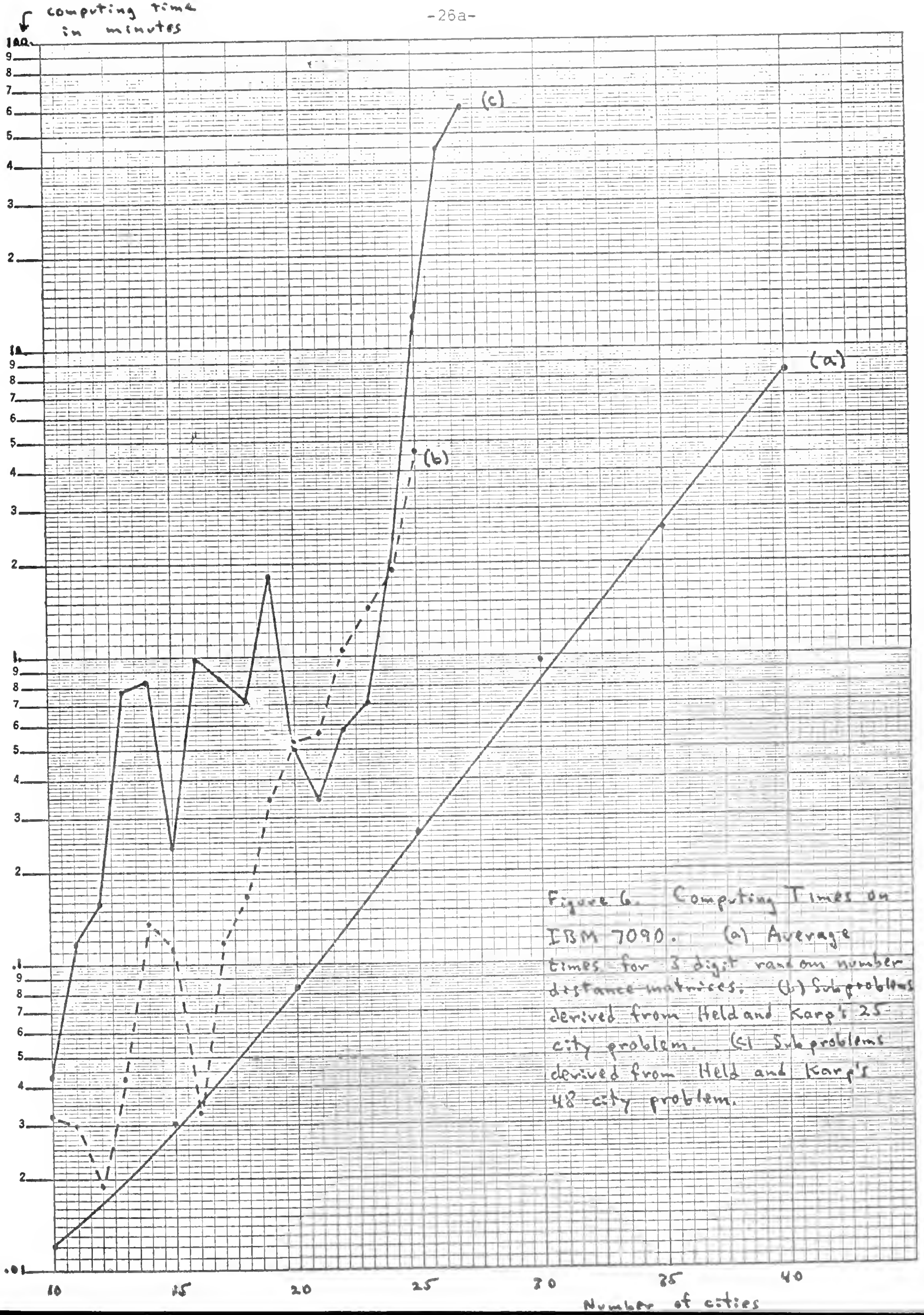

Figure 6. Computing Times on IBM 7090. (a) Average times for 3 digit random number distance matrices. (b) Subproblems derived from Held and Karp's 25 city problem. (c) Subproblems derived from Held and Karp's 48 city problem.

References

1. M. M. Flood, "The Traveling Salesman Problem," *Opns. Res.*, **4**, 61-75, (1956).

2. R. H. Gonzales, "Solution of the Traveling Salesman Problem by Dynamic Programming on the Hypercube," Interim Technical Report No. 18, OR Center, M. I. T., 1962.

3. M. Held and R. M. Karp, "A Dynamic Programming Approach to Sequencing Problems," *J. Soc. Indust. and Appl. Math.*, **10**, 196-210, (1962).

4. M. J. Rossman, R. J. Twery, and F. D. Stone, "A Solution to the Traveling Salesman Problem by Combinatorial Programming," mimeographed.

5. M. J. Rossman and R. J. Twery, "Combinatorial Programming", presented at 6th Annual ORJA meeting, May 1958, mimeographed.

6. W. L. Eastman, "Linear Programming with Pattern Constraints," Ph.D. dissertation, Harvard University, July 1958. Also, in augmented form; Report No. BL-20 The Computation Laboratory, Harvard University, July 1958.

7. G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson, "Solution of a Large Scale Traveling Salesman Problem," *Opns. Res.*, **2**, 393-410 (1954).

8. G. A. Croes, "A Method for Solving Traveling Salesman Problems", *Opns. Res.*, **6**, 791-814 (1958).

07-63

08-63

09-63

10-63

11-63

12-63

13-63

14-63